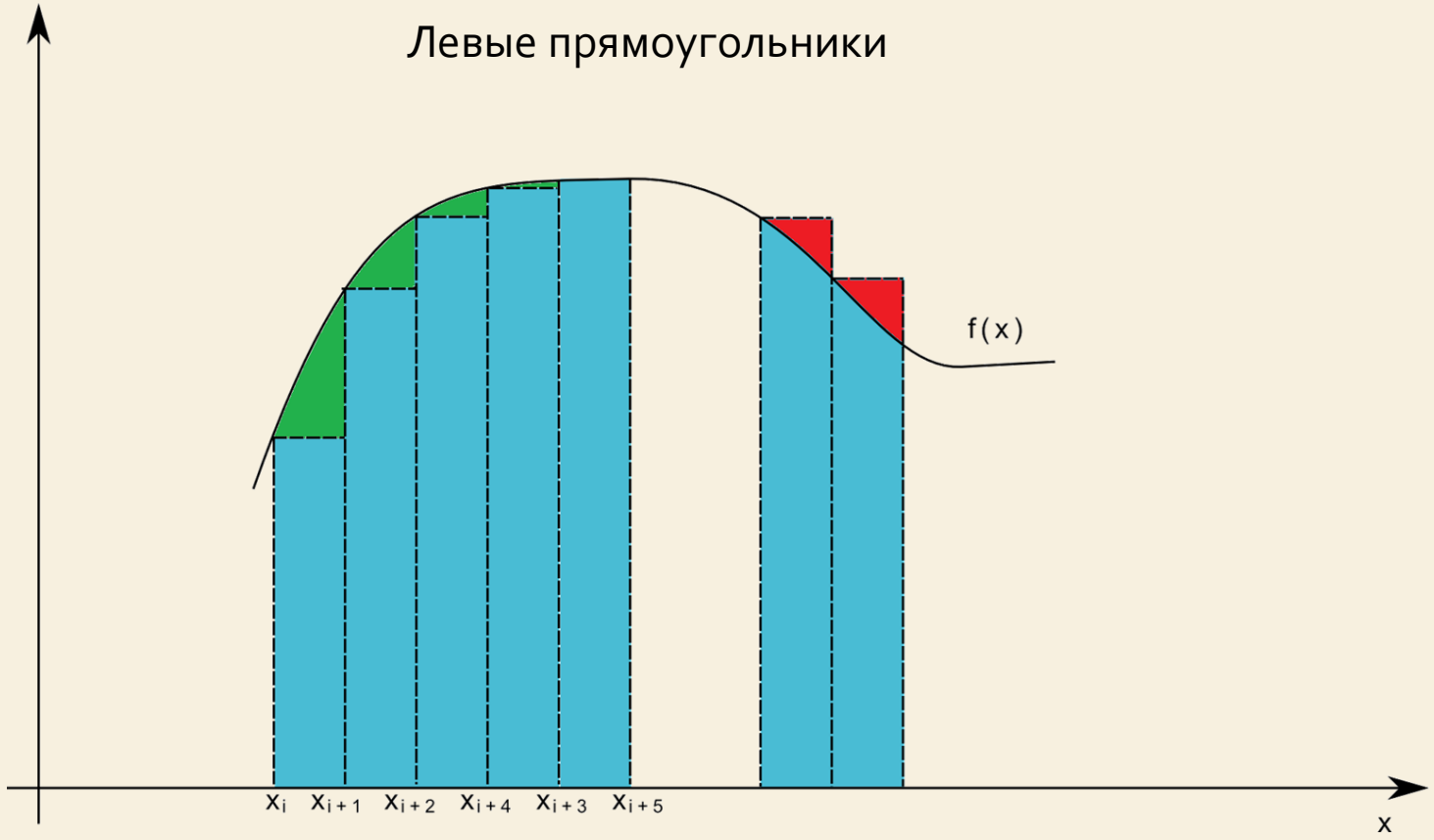
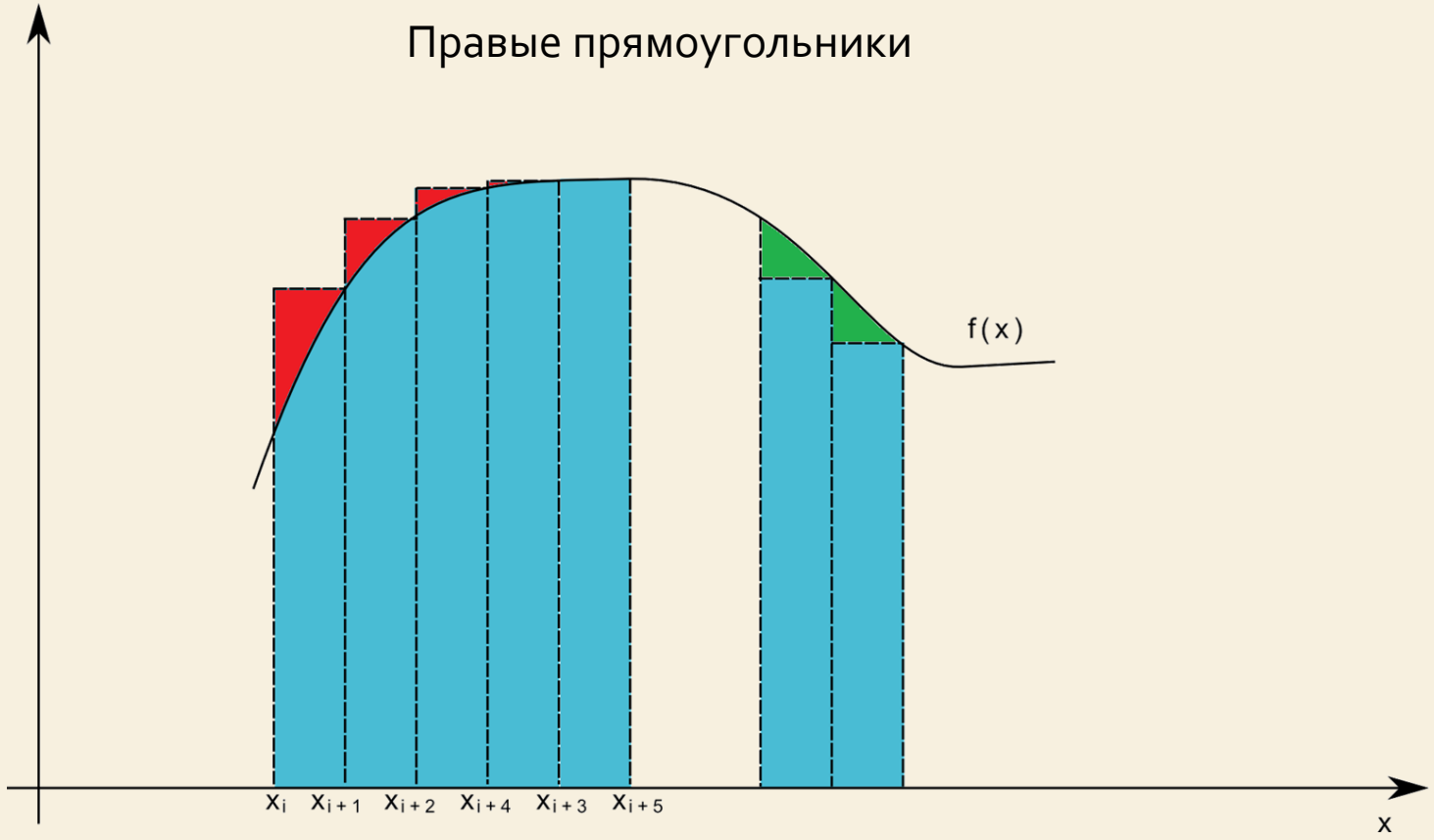
СИСТЕМЫ ТЕХНИЧЕСКОГО ЗРЕНИЯ

ИНТЕГРИРОВАНИЕ ФУНКЦИЙ. ИНТЕГРИРОВАНИЕ ОДУ.

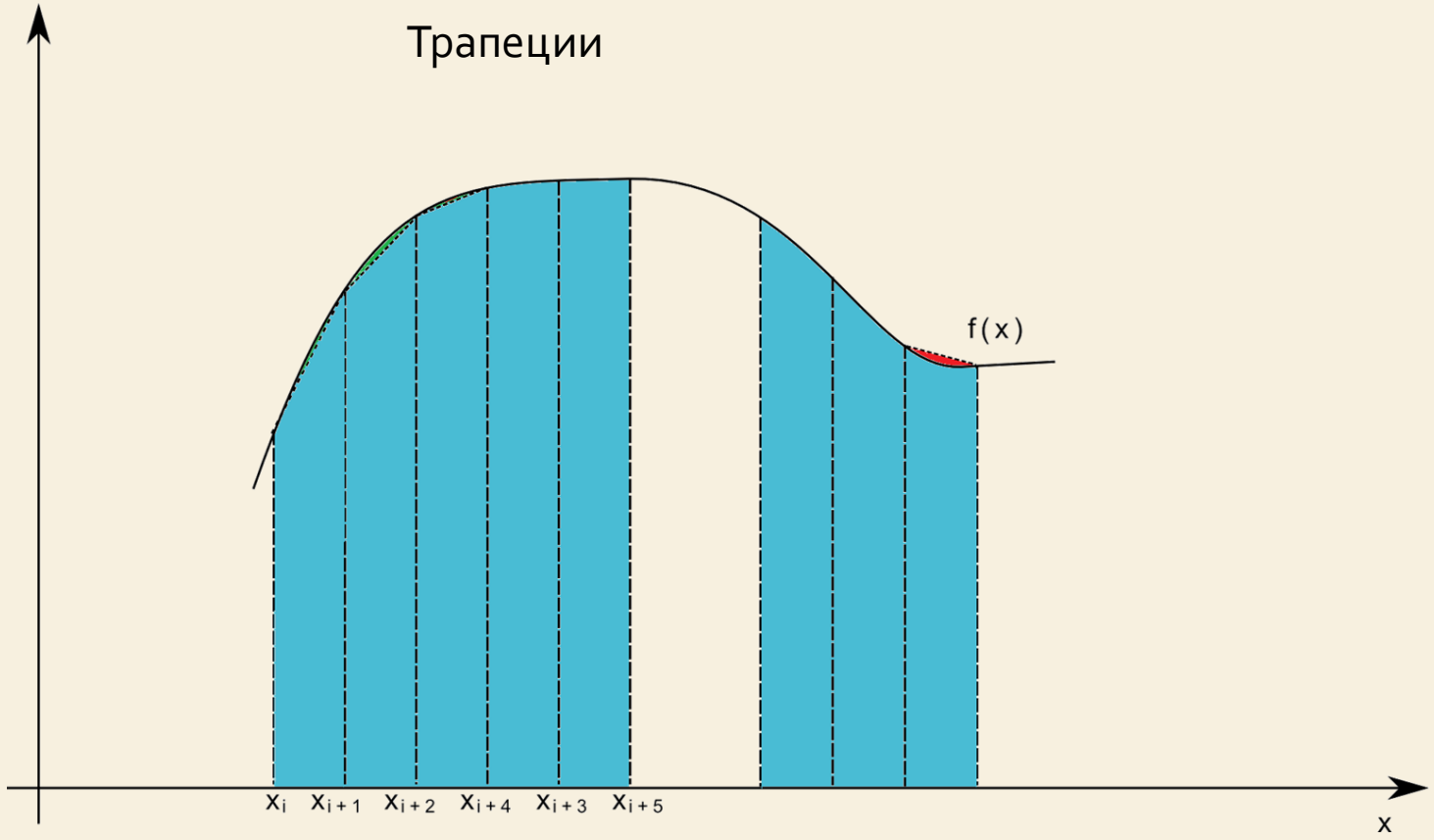
ИНТЕГРИРОВАНИЕ ФУНКЦИЙ



ИНТЕГРИРОВАНИЕ ФУНКЦИЙ



ЧУТЬ СЛОЖНЕЕ - ТРАПЕЦИИ



В ВИДЕ ФОРМУЛ

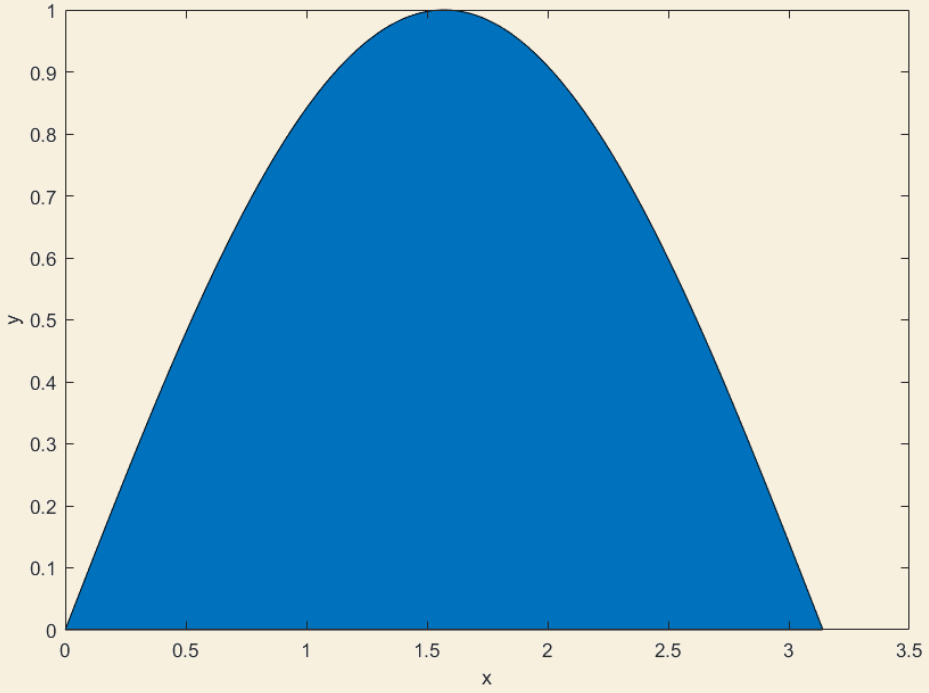
- Прямоугольники:
  - Левые $S = \sum_i^n f(x_i) \cdot h$
  - Правые $S = \sum_i^n f(x_{i+1}) \cdot h$
  - Трапеции $S = \sum_i^n (f(x_{i+1}) + f(x_i)) \cdot \frac{h}{2}$
  - Симпсона

$$S = \sum_{i=1,2}^{n-1} [f(x_{i-1}) + 4f(x_i) + f(x_{i+1})] \cdot \frac{h}{3}$$

ПРИМЕР ИНТЕГРИРОВАНИЯ

- Мы возьмём функцию $y = \sin(x)$ и попробуем её проинтегрировать на отрезке $x \in [0; \pi]$
- Аналитический результат нам известен:

$$\int_0^{\pi} \sin(x)\,dx = -\cos(x)\Big|_0^{\pi} = 1 - (-1) = 2$$



```cpp
#include <iostream>
#include <fstream>
#include <vector>
#define _USE_PREDEFINED_CONSTANT
#include <cmath>
#include <limits>

using namespace std;

void MakeData(vector<double>& x, vector<double>& y);
double IntegrRectLeft (const vector<double>& x, const vector<double>& y);
double IntegrRectRight(const vector<double>& x, const vector<double>& y);
double IntegrTrap     (const vector<double>& x, const vector<double>& y);

int main()
{
    vector<double> x, y;
    MakeData(x, y);
    cout << "Real value " << cos(0)-cos(M_PI) << endl;
    cout.precision(15);
    cout << "Left rectangles  " << IntegrRectLeft(x,y) << endl;
    cout << "Right rectangles " << IntegrRectRight(x,y) << endl;
    cout << "Trapezoid        " << IntegrTrap(x,y) << endl;
    return 0;
}

void MakeData(vector<double>& x, vector<double>& y)
{
    double step = 0.001;
    for (float i = 0; i <= M_PI+step/2; i+=step)
    {
        x.push_back(i);
        y.push_back(sin(i));
    }
}
```

СРАВНЕНИЕ

Мне пришлось поставить точность выше, чтобы разница была визуально заметной

```cpp
double IntegrRectLeft(const vector<double>& x, const vector<double>& y)
{
    double summ = 0;
    for (int i=0; (i<y.size()-1)&&(i<x.size()-1); i++)
    {
        double delta = x[i+1]-x[i]; //  Шаг интегрирования
        double value = y[i];
        summ += value*delta;
    }
    return summ;
}
double IntegrRectRight(const vector<double>& x, const vector<double>& y)
{
    double summ = 0;
    double delta = x[1]-x[0];       //  На самом деле он у нас постоянный
    for (int i=0; (i<y.size()-1)&&(i<x.size()-1); i++)
    {
        double value = y[i+1];
        summ += value*delta;
    }
    return summ;

}

double IntegrTrap(const vector<double>& x, const vector<double>& y)
{
    double summ = 0;
    for (int i=0; (i<y.size()-1)&&(i<x.size()-1); i++)
    {
        double delta = x[i+1]-x[i];
        double value = y[i+1]+y[i];
        summ += value*delta/2;
    }
    return summ;
}
```

РЕАЛИЗАЦИЯ МЕТОДОВ

В большинстве ваших задач (и в курсах обработки датчиков) величина шага может считаться постоянной – частота работы СУ фиксирована

ПОЧТИ ЛОГИЧНЫЙ РЕЗУЛЬТАТ

ВЫВОД ПРОГРАММЫ

- Real value 2
- Left rectangles 1.99999991713251
- Right rectangles 2.00000275144512
- Trapezoid 1.99999976702904

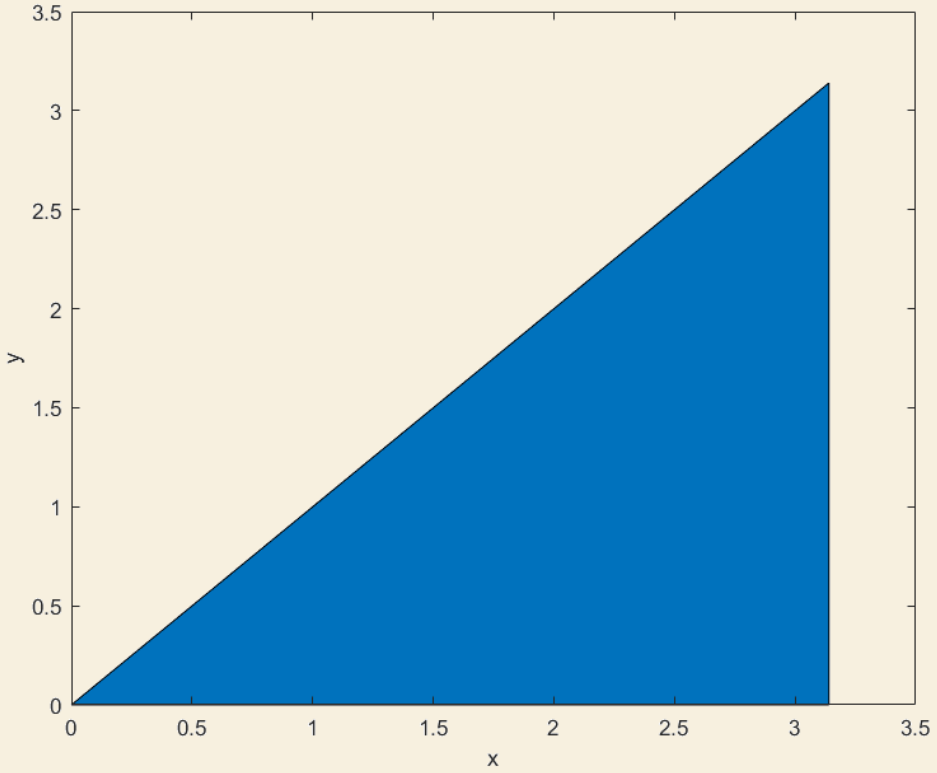
- Можно заметить, что результаты близки
- Но метод левых прямоугольников оказался точнее
- Почему?

ЕЩЁ БОЛЕЕ ПРОСТАЯ (НО ПОКАЗАТЕЛЬНАЯ) ФУНКЦИЯ

- Теперь попробуем рассмотреть функцию $y = x$ на отрезке $x \in [0; 1]$
- Аналитический результат ещё проще

$$\int_0^1 x dx = \frac{x^2}{2}\Big|_0^1 = \frac{1}{2} - 0 = \frac{1}{2}$$

- На такой функции метод трапеций должен давать **точный** результат
- Теперь я попробую посчитать с разной величиной шага

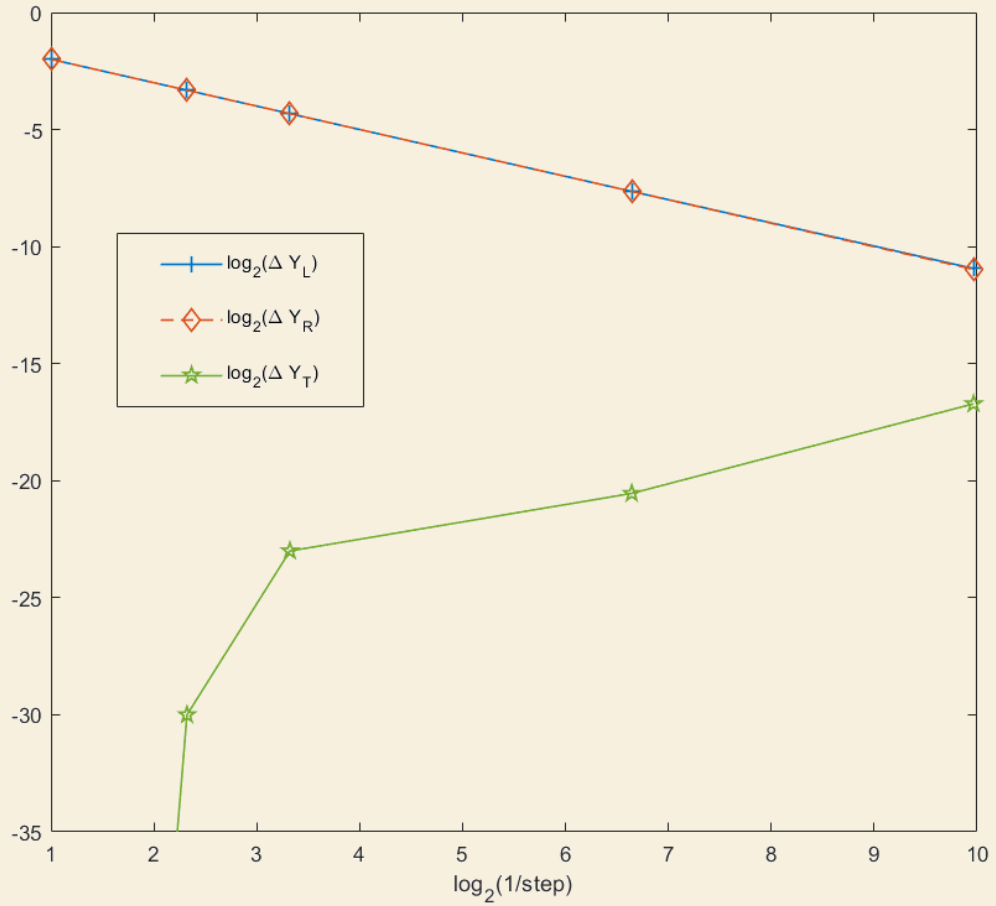


РЕЗУЛЬТАТЫ

Шаг	Левые прямоугольники		Правые прямоугольники		Трапеции	
	Значение	Ошибка	Значение	Ошибка	Значение	Ошибка
0,5	0.25	0.25	0.75	0.25	0.5	0
0,2	0.4	0.1	0.6	0.1	0.5	0
0,1	0.450000107	0.04999989	0.550000046	0.050000046	0.500000119	0.000000119
0,01	0.494999350	0.005000649	0.504999764	0.004999764	0.499999344	0.000000655
0,001	0.499490711	0.000509288	0.500496684	0.000496684	0.499990701	0.000009298

СНОВА О ПОГРЕШНОСТЯХ

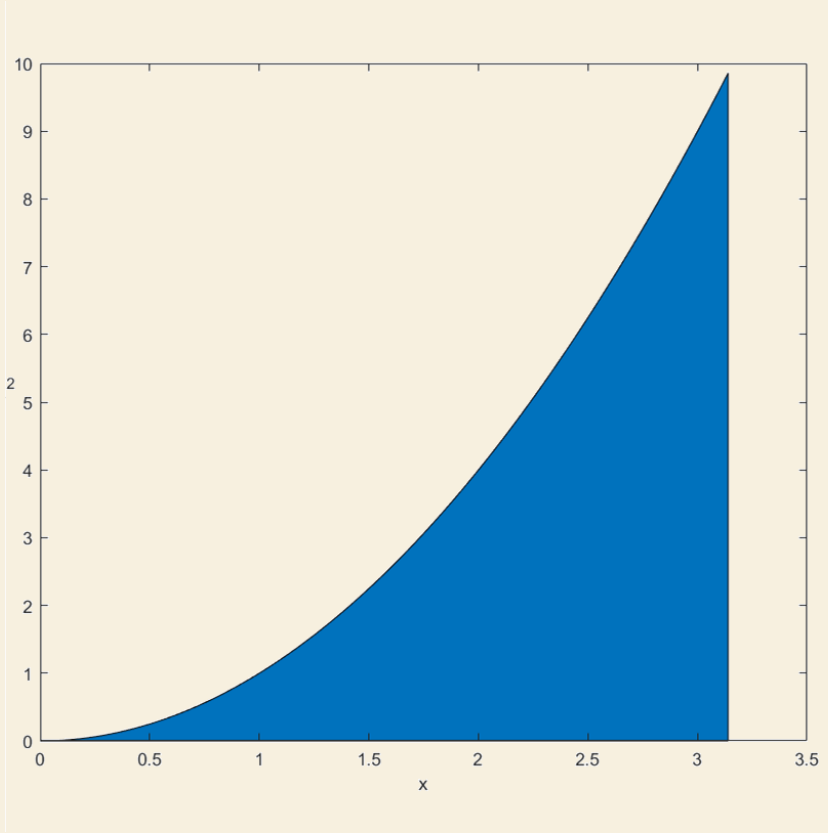
- Для обоих методов прямоугольников погрешность уменьшается
- Для изначально точного метода трапеций - растёт
- Эффект накопления вычислительной ошибки



МЕТОД СИМПСОНА – АППРОКСИМАЦИЯ ПАРАБОЛАМИ

- В методе прямоугольников мы аппроксимировали константой, в трапециях – прямой
- Метод Симпсона – вычисление площади под параболой
- Должен быть точен на квадратичных функциях
- Значит, мы рассмотреть функцию $y = x^2$ на отрезке $x \in [0; 1]$
- Аналитический результат ещё проще

$$\int_0^1 x^2 dx = \frac{x^3}{3}\Big|_0^1 = \frac{1}{3} - 0 = \frac{1}{3}$$

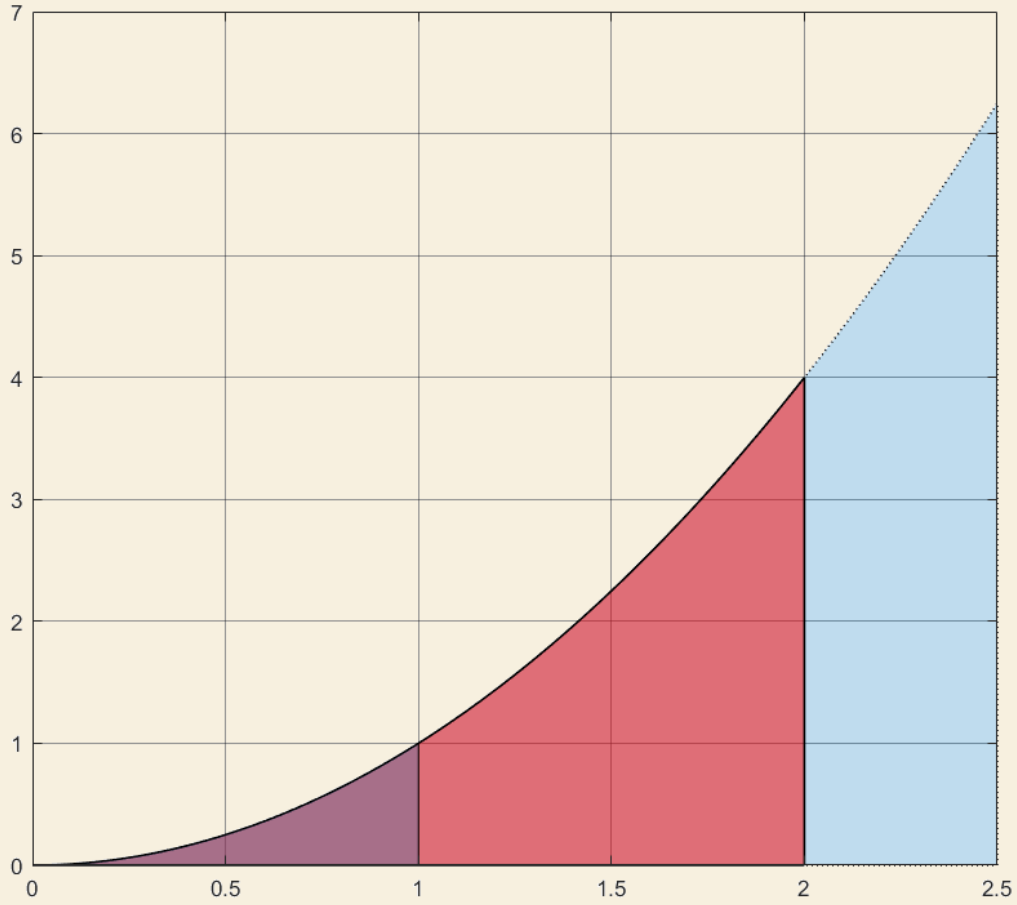


```
void MakeData(vector<double>& x, vector<double>& y)
{
    double step = 0.1;
    for (float i = 0; i <=1+step/2; i+=step)
    {
        x.push_back(i);
        y.push_back(i*i);
    }
}
double IntegrSimpson(const vector<double>& x, const vector<double>& y)
{
    double summ = 0;
    for (int i=0; (i<y.size()-2)&&(i<x.size()-2); i+=2)
    {
        double delta = (x[i+2]-x[i])/2;
        double value = (y[i+2]+4*y[i+1]+y[i])/3;
        summ += value*delta;
    }
    if (y.size()%2 == 0)
    {   //  Один "лишний" интервал посчитаем методом трапеций
        summ += (x[x.size()-1]-x[x.size()-2])*(y[y.size()-1]+y[y.size()-2])/2;
    }
    return summ;
}
```

МЕТОД СИМПСОНА

Сложность с применением – для построения параболы нужно 3 точки

Разбиваем интервал интегрирования на подотрезки из 2



НЕЧЁТНОЕ КОЛИЧЕСТВО ОТРЕЗКОВ

У нас пять отрезков – мы можем использовать метод Симпсона на первых двух парах

МЕТОД СИМПСОНА - РЕЗУЛЬТАТ

- Real value 0.333333
- Left rectangles 0.285000099167239
- Right rectangles 0.385000069066886
- Trapezoid 0.335000119805350
- Simpson 0.333333451151862

- Метод точнее всех
- Однако уже здесь видно появление вычислительной ошибки

ТЕОРЕТИЧЕСКИЕ ОЦЕНКИ ТОЧНОСТЕЙ

- Для метода «левых» и «правых» прямоугольников **при постоянном шаге**

$$E(f) = \frac{\max|f'(x)|}{2}(b-a)h$$

- Для метода трапеций **при постоянном шаге**

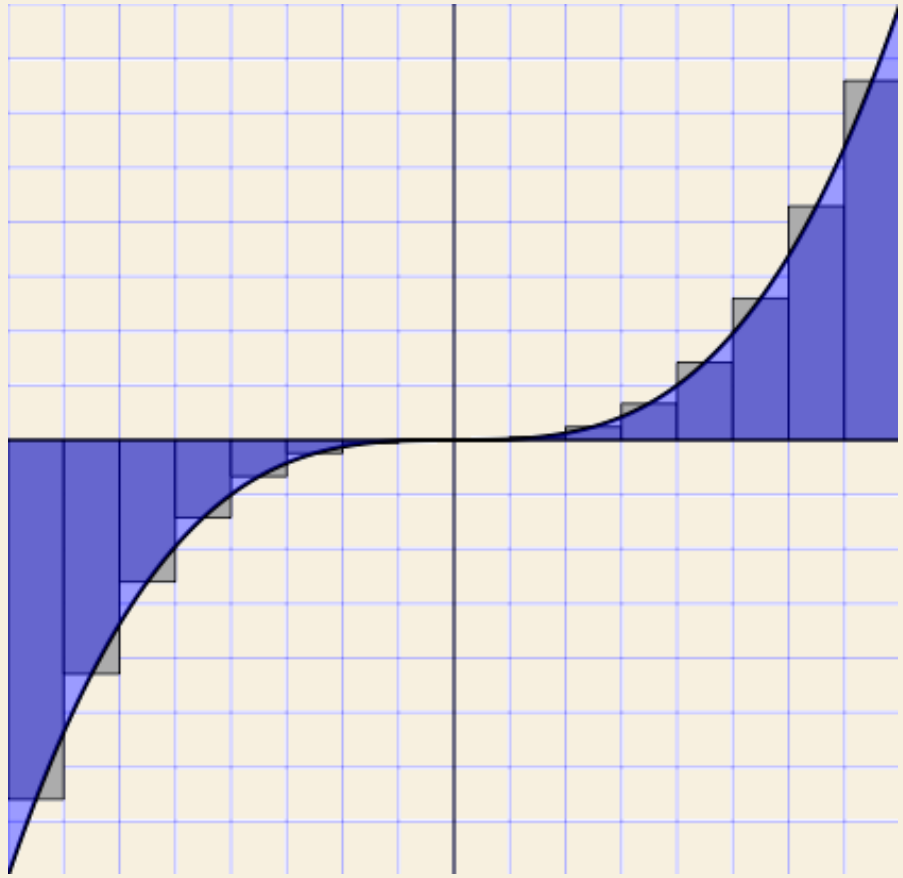
$$E(f) = \frac{\max|f''(x)|}{12}(b-a)h^2$$

- Для метода Симпсона **при постоянном шаге**

$$E(f) = \frac{\max|f'''(x)|}{288}(b-a)h^3$$

МЕТОД ЦЕНТРАЛЬНЫХ ПРЯМОУГОЛЬНИКОВ

- Похож на остальные методы прямоугольников, но точка берётся в центре
- Точен для линейных функций – как метод трапеций
- Не применим, если функция задана в виде набора значений



МЕТОД (ИНТЕГРИРОВАНИЯ) ГАУССА

- Общая идея – если правильно выбирать точки интегрирования, можно по малому числу точек построить интеграл, точный для *полиномиальной* функции порядка *n*
- Например, для того чтобы точно проинтегрировать функции нулевого и первого порядка – достаточно одной центральной точки
- Для интегрирования второго и третьего – всего двух точек, и так далее
- Самый точный, но не применим, если нельзя выбрать точки (например, функция задана в виде таблицы)
- Смотреть в учебнике

ИНТЕГРИРОВАНИЕ ОДУ

- Обыкновенное дифференциальное уравнение с начальными условиями

$$\frac{du(x)}{dx} = f(x,u), \qquad u(x_0) = u_0$$

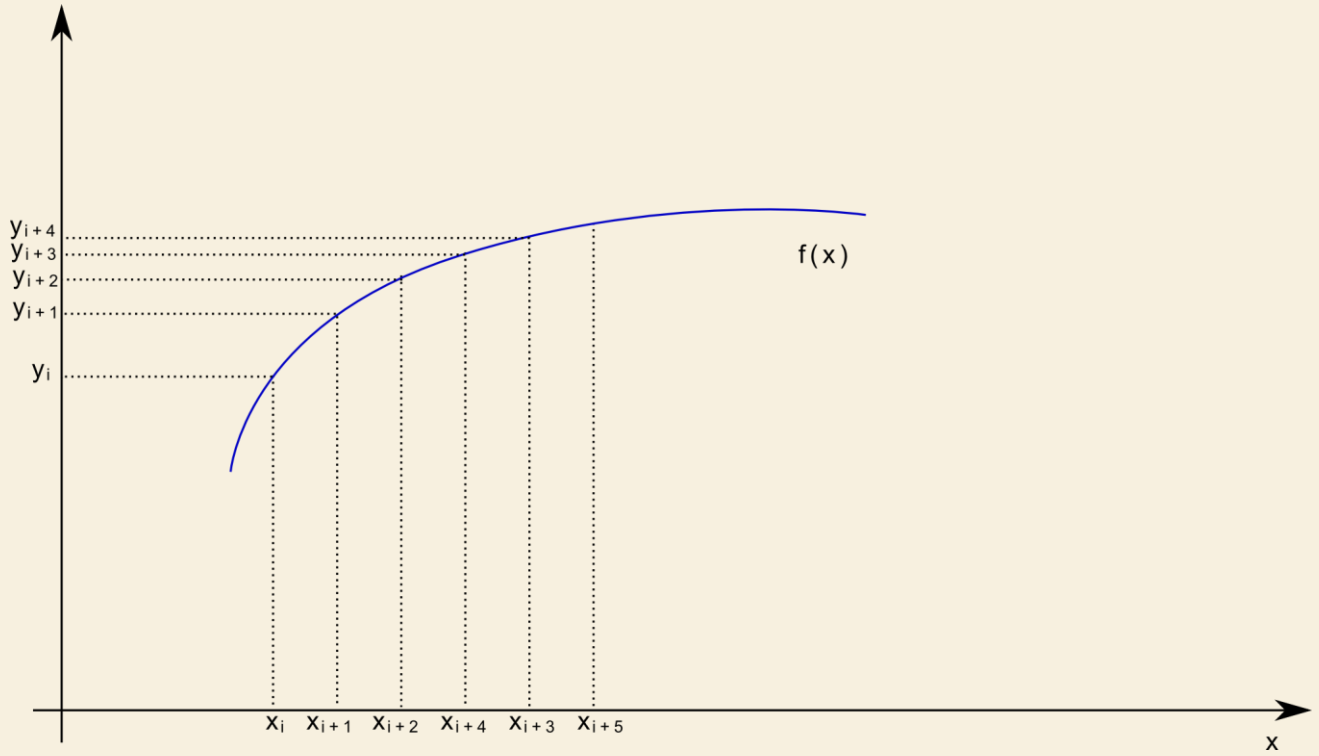
- У нас возникает необходимость его численного решения – например, для построения модели
- Попробуем его дискретизировать:

$$x_i = [x_0, x_1, x_2, \dots, x_n]$$

$$u(x) \to y(x_i) = y_i$$

$$\frac{du(x)}{dx} = \lim_{\Delta x \to 0} \frac{\Delta u}{\Delta x} \to \frac{\Delta y_i}{\Delta x_i}$$

ИНТЕГРИРОВАНИЕ ОДУ



УРАВНЕНИЕ ПОСЛЕ ДИСКРЕТИЗАЦИИ

После дискретизации

$$\frac{\Delta y_i}{\Delta x_i} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

Тогда, если шаг постоянен

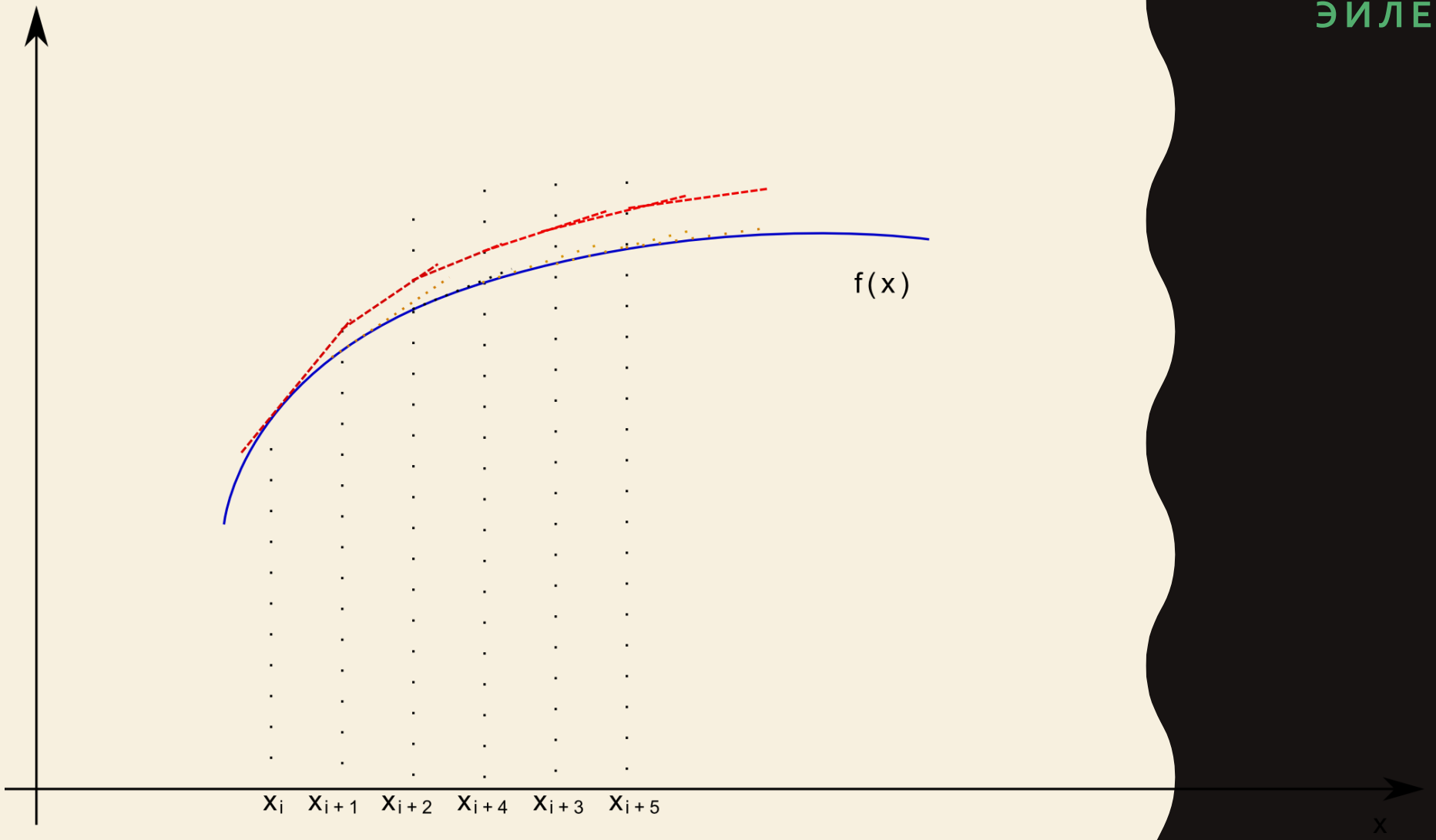
$$x_{i+1} - x_i = h_i = h$$

наше исходное уравнение можно записать

$$\frac{y_{i+1} - y_i}{h} = f(x_i, y_i) \quad \Rightarrow \quad y_{i+1} = y_i + h \cdot f(x_i, y_i), \qquad y_0 = u_0$$

ЯВНАЯ СХЕМА ЭЙЛЕРА

$$y_{i+1} = y_i + h \cdot f(x_i, y_i), \qquad y_0 = u_0$$



ИТЕРАЦИОННЫЙ ПРОЦЕСС

- Для явной схемы получаем

$$y_1 = y_0 + h \cdot f(x_0, y_0)$$
$$y_2 = y_1 + h \cdot f(x_1, y_1)$$
$$\dots$$

- Аналогично можно получить неявную схему

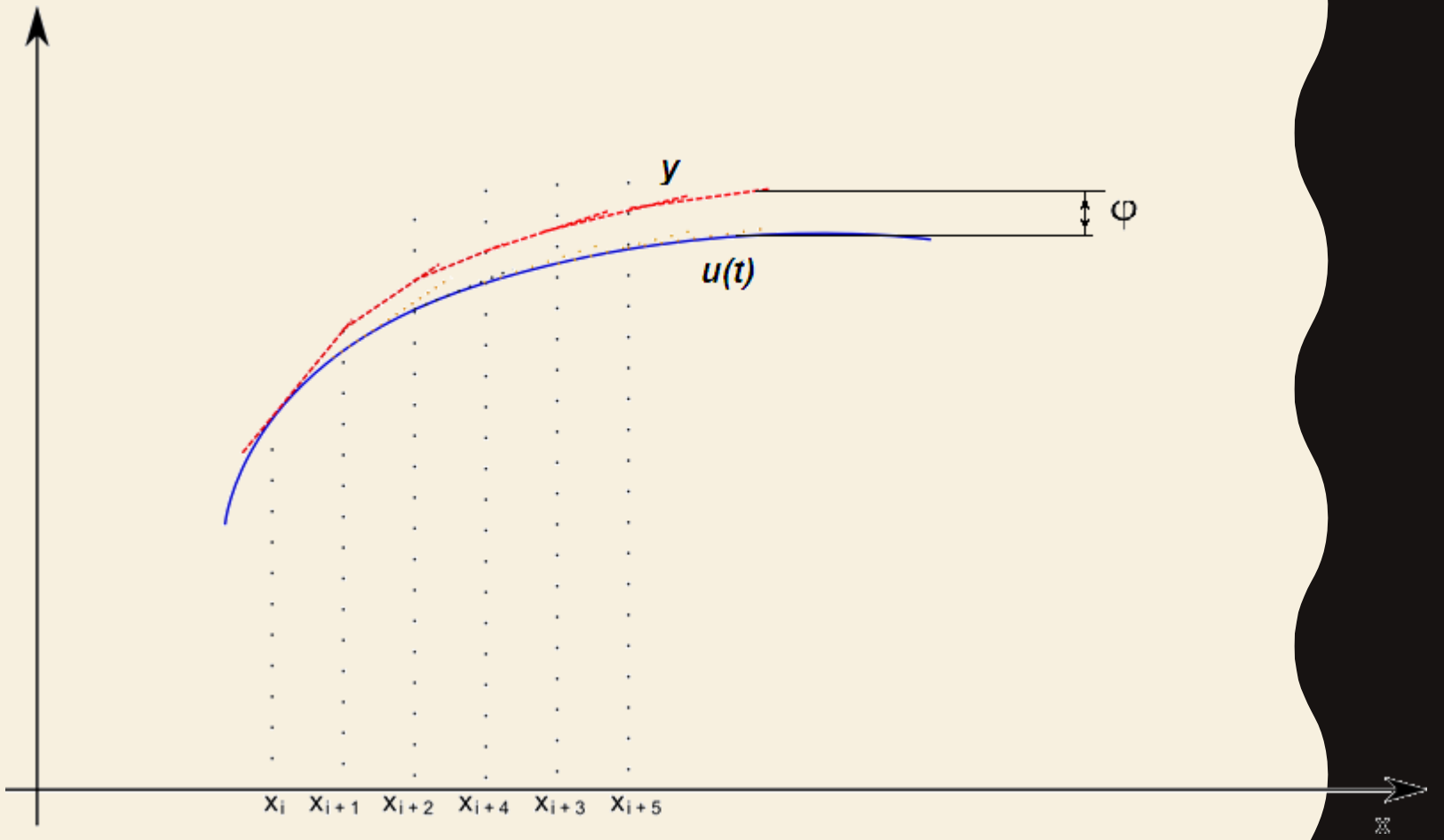
$$y_{i+1} = y_i + h \cdot f(x_{i+1}, y_{i+1}), y_0 = u_0$$

- Или полуявную схему

$$y_{i+1} = y_i + \frac{h}{2} \cdot [f(x_i, y_i) + f(x_{i+1}, y_{i+1})], y_0 = u_0$$

- В этом случае для поиска y_{i+1} на каждом шаге нужно решать нелинейное уравнение (например, методом половинного деления)

$$\varphi_i = u(t_i) - y_i \ - \text{погрешность}$$



НАСКОЛЬКО ТОЧНОЕ РЕШЕНИЕ МЫ ПОЛУЧИЛИ?

ПОГРЕШНОСТЬ

Насколько точно мы получаем решение?

Для явной схемы:

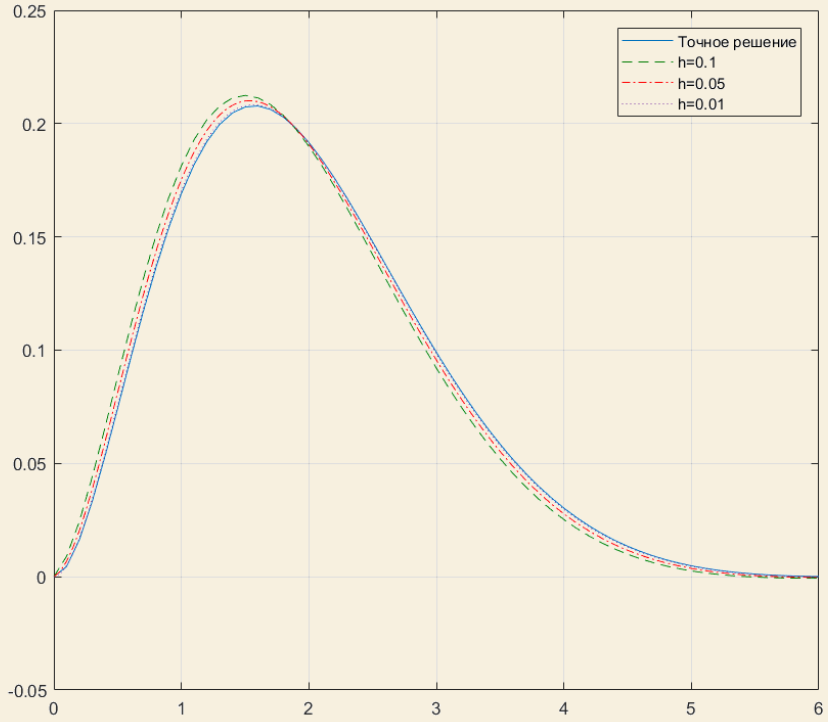
$$\frac{y_{i+1} - y_i}{h} = \mathrm{y}_\mathrm{i}' + \frac{y_i''}{2h^2} + \cdots$$

Поэтому

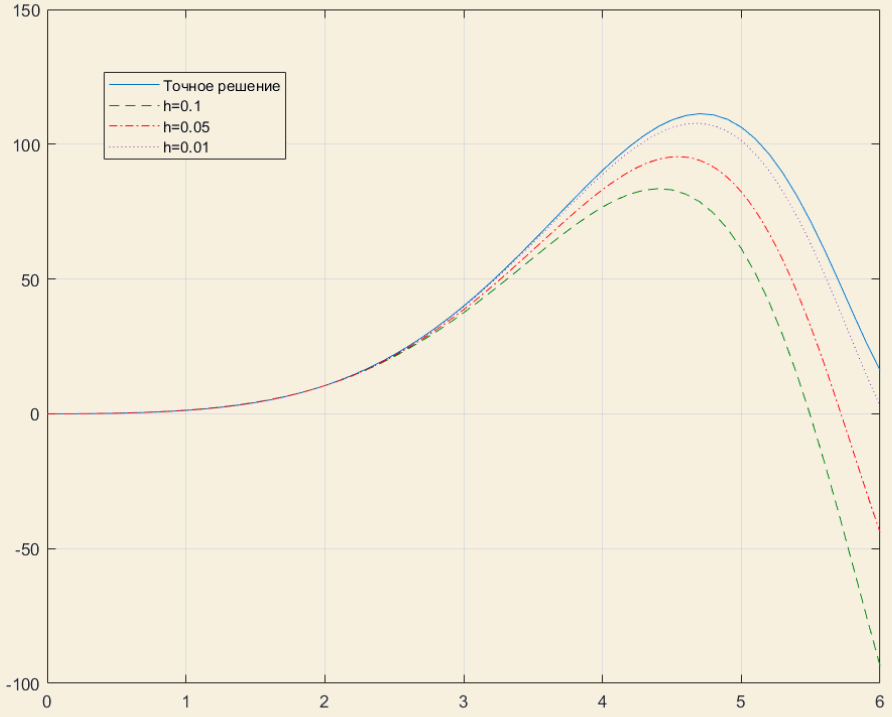
$$\max_{i \in N} |u(x_i) - y_i| \leq \max_{i \in N} |y_i''| \cdot \frac{\mathrm{h}}{2} = \mathrm{Mh}$$

ПОГРЕШНОСТЬ ЗАВИСИТ ОТ РЕШЕНИЯ

$$\frac{dy}{dt} = -y + sin(t) \cdot e^{-t}$$



$$\frac{dy}{dt} = y + sin(t) \cdot e^{t}$$



ПОГРЕШНОСТЬ

Аналогично для полуявной схемы:

$$\frac{y_{i+1} - y_{i-1}}{2h} = \mathrm{y}_\mathrm{i}' + \frac{y_i'''}{6h^3} + \cdots$$

Поэтому

$$\max_{i \in N} |u(x_i) - y_i| \leq \max_{i \in N} |y_i'''| \cdot \frac{\mathrm{h}^2}{6} = \mathrm{Mh}^2$$

МЕТОДЫ РУНГЕ-КУТТЫ

$$\frac{du}{dx} = f(x, u)$$

Общая идея – давайте попробуем посчитать определённый интеграл:

$$u(x_{i+1}) = u(x_i) + \int_{x_i}^{x_{i+1}} f(x, u)dx$$

Определённая сложность в том, что он обычно не вычисляем

МЕТОДЫ РУНГЕ-КУТТЫ

И снова дискретизируем:

$$x_i = [x_0, x_1, \dots, x_n]$$
$$y_i = [u(x_0), u(x_1), \dots, u(x_n)]$$

Интеграл заменим квадратурной формулой:

$$y_{i+1} = y_i + h \sum_{k=1}^{m} c_k f\left(x_i^{(k)}, y\left(x_i^{(k)}\right)\right)$$

Например, при m=2 получим метод Эйлера

МЕТОДЫ РУНГЕ-КУТТЫ

Схема Рунге-Кутты 4-го порядка точности

$$y_{n+1} = y_n + h \cdot k_n, \qquad k_n = \frac{1}{6}\left(k_n^{(1)} + 2k_n^{(2)} + 2k_n^{(3)} + k_n^{(4)}\right)$$

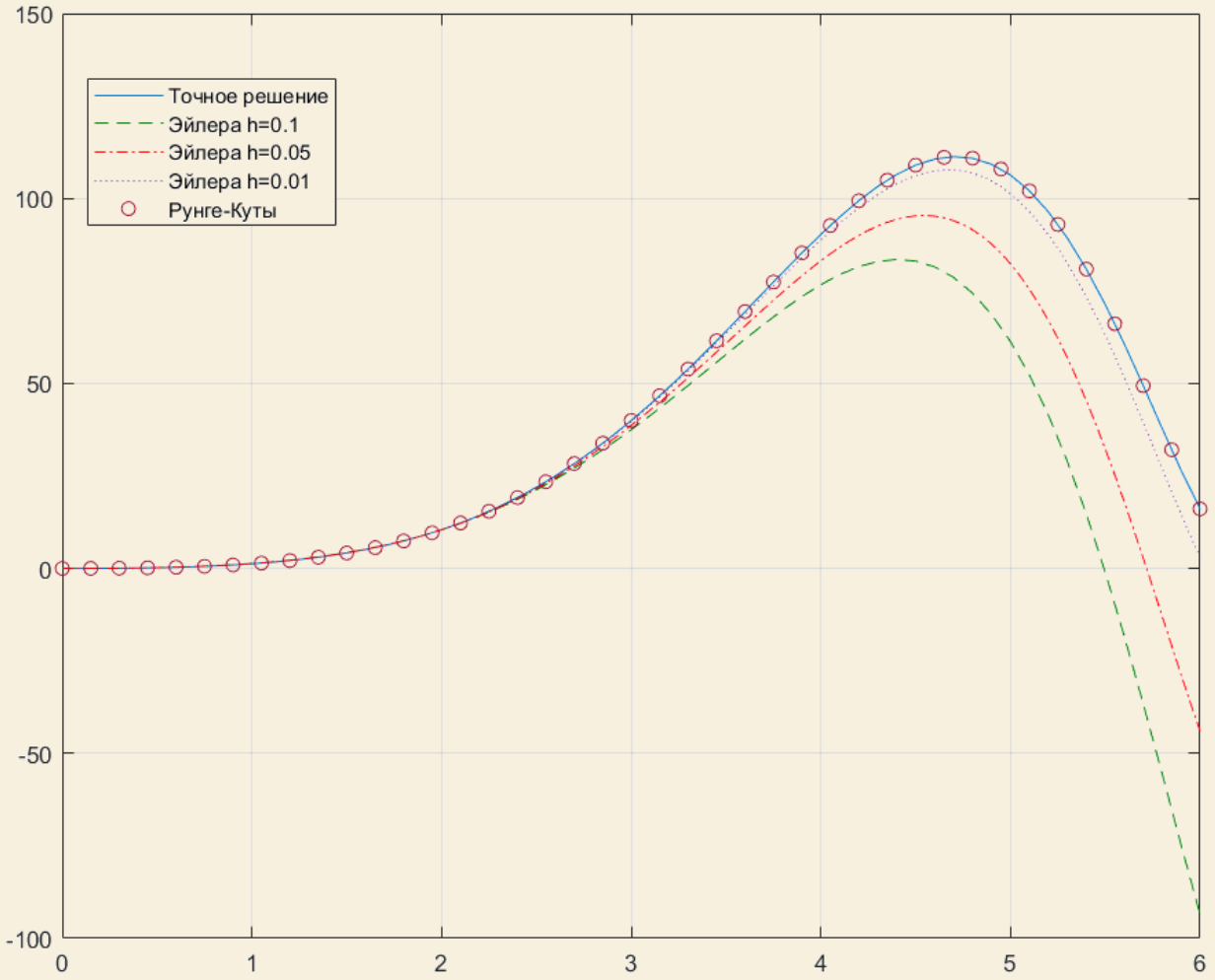
$$k_n^{(1)} = f(x_n, y_n), \qquad k_n^{(2)} = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_n^{(1)}\right),$$

$$k_n^{(3)} = f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_n^{(2)}\right), \qquad k_n^{(4)} = f\left(x_n + h, y_n + hk_n^{(3)}\right),$$

В MATLAB – ode45() – с выбором шага

MATLAB

```
tspan = [0 6];
y0 = 0;
[t,y] = ode45(@(t,y) y+sin(t)*exp(t), tspan, y0);
plot(t, y, 'o')
```



НЕЯВНЫЕ МЕТОДЫ РУНГЕ-КУТТЫ

- Явные методы не подходят для решения «жёстких» задач
- Неявные методы – лучше, в силу большей устойчивости
- При этом задача решается итерационно
- Пример – неявный метод Эйлера (он же метод Рунге 1 порядка):

$$\tilde{y}_{n+1} = y_n + h \cdot f(x_n, y_n)$$

$$y_{n+1} = y_n + h\frac{f(x_n, y_n) + f(x_{n+1}, \tilde{y}_{n+1})}{2}$$

МЕТОД РУНГЕ-КУТТЫ 4 ПОРЯДКА

```cpp
#include <iostream>
#define USE_MATH_DEFINES
#include <cmath>
#include <vector>
using std::cout;
using std::endl;

double f (double x, double y)
{
   return (-sin(x));
}

double runge4(double t, double y0, double step);

int main(int argc, char *argv[])
{
   double initVal = 1;
   double step = 0.1;
   std::vector<double> time, y;
   time.push_back(0);
   y.push_back(initVal);
   for(double t = step; t <M_PI; t+=step)
   {
      double dy = runge4(time.back(), y.back(), step);
      time.push_back(t);
      y.push_back(dy);
   }
   for (int i=0; i<time.size(); i++)
   {
      cout << "Cos(t)=" << cos(time[i]) << " y(t)=" << y[i] << endl;
   }
   return 0;
}
```

МЕТОД РУНГЕ-КУТТЫ 4 ПОРЯДКА

```
double runge4(double t, double y0, double step)
{
    double k1, k2, k3, k4;
    k1 = f(t, y0);
    k2 = f(t+step/2, y0+step*k1/2);
    k3 = f(t+step/2, y0+step*k2/2);
    k4 = f(t+step, y0+step*k3);
    double y = y0 +(k1+2*k2+2*k3+k4)*step/6;
    return y;
}
```

```
Cos(x)=1 val=1
Cos(x)=0.995004 val=0.985062
Cos(x)=0.980067 val=0.960332
Cos(x)=0.955336 val=0.926057
Cos(x)=0.921061 val=0.882578
Cos(x)=0.877583 val=0.830331
Cos(x)=0.825336 val=0.769838
Cos(x)=0.764842 val=0.701703
Cos(x)=0.696707 val=0.626606
Cos(x)=0.62161 val=0.545298
Cos(x)=0.540302 val=0.458592
Cos(x)=0.453596 val=0.367354
Cos(x)=0.362358 val=0.272495
Cos(x)=0.267499 val=0.174963
Cos(x)=0.169967 val=0.075733
Cos(x)=0.0707372 val=-0.0242037
Cos(x)=-0.0291995 val=-0.123849
Cos(x)=-0.128844 val=-0.222206
Cos(x)=-0.227202 val=-0.318294
Cos(x)=-0.32329 val=-0.411151
Cos(x)=-0.416147 val=-0.49985
Cos(x)=-0.504846 val=-0.583505
Cos(x)=-0.588501 val=-0.66128
Cos(x)=-0.666276 val=-0.732398
Cos(x)=-0.737394 val=-0.796148
Cos(x)=-0.801144 val=-0.851893
Cos(x)=-0.856889 val=-0.899076
Cos(x)=-0.904072 val=-0.937227
Cos(x)=-0.942222 val=-0.965962
Cos(x)=-0.970958 val=-0.984997
Cos(x)=-0.989992 val=-0.994139
Cos(x)=-0.999135 val=-0.993299
Press <RETURN> to close this window...
```

КАК РЕШИТЬ БОЛЕЕ СЛОЖНОЕ УРАВНЕНИЕ?

- Мы с вами выяснили, как решить уравнение

$$\frac{du}{dt} = f(t,u), u(t_0) = u_0$$

А как нам быть, если уравнение несколько сложнее, например:

$$(m + \lambda_{11}) \cdot \ddot{x} + C_1 \dot{x}|\dot{x}| + C_2 \dot{x} - F = 0, \qquad x(t_0) = u_0, v(t_0) = 0$$

Метод Рунге-Кутты – для уравнений первого порядка

СИСТЕМА УРАВНЕНИЕ

- Перейдём к системе уравнений:
- $$\begin{cases} \dot{x} = v, \\ (m + \lambda_{11}) \cdot \dot{v} + C_1 v|v| + C_2 v - F = 0 \end{cases}$$
- С начальными условиями $x(t_0) = u_0, v(t_0) = 0$
- К чему это приведёт с точки зрения кода?

ФУНКЦИИ ПРАВЫХ ЧАСТЕЙ

```
#include <iostream>
#include <vector>

using namespace std;

double B1 = 1,
B2 = 1,
F = 10;

double f1(double x, double v, double t)
{
        return v;
}
double f2(double x, double v, double t)
{
        return (-x);
        // return (-B1*v*v - B2*v + F);
}
```

КЛАСС-РЕШАТЕЛЬ

```
class Runge4Solver
{
public:
        Runge4Solver(double x, double v, double t)
        {
                m_x.push_back(x);
                m_v.push_back(v);
                m_time.push_back(t);
        }
        double x() const {return m_x.back();}
        double v() const {return m_v.back();}
        double t() const {return m_time.back();}
        void CalcStep()
        {
                ...
        }
private:
        std::vector<double> m_time, m_x, m_v;
        double k_x[4], k_v[4];
        double m_step = 0.01;
};
```

CALCSTEP

```
void CalcStep()
{
        double x_prev = m_x.back(),
        v_prev = m_v.back(),
        t_prev = m_time.back();
        k_x[0] = f1(x_prev, v_prev, t_prev);
        k_v[0] = f2(x_prev, v_prev, t_prev);
        k_x[1] = f1(x_prev+k_x[0]*m_step/2, v_prev+k_v[0]*m_step/2, t_prev+m_step/2);
        k_v[1] = f2(x_prev+k_x[0]*m_step/2, v_prev+k_v[0]*m_step/2, t_prev+m_step/2);
        k_x[2] = f1(x_prev+k_x[1]*m_step/2, v_prev+k_v[1]*m_step/2, t_prev+m_step/2);
        k_v[2] = f2(x_prev+k_x[1]*m_step/2, v_prev+k_v[1]*m_step/2, t_prev+m_step/2);
        k_x[3] = f1(x_prev+k_x[2]*m_step, v_prev+k_v[2]*m_step, t_prev+m_step);
        k_v[3] = f2(x_prev+k_x[2]*m_step, v_prev+k_v[2]*m_step, t_prev+m_step);
        m_x.push_back(x_prev+m_step*(k_x[0]+2*k_x[1]+2*k_x[2]+k_x[3])/6.0);
        m_v.push_back(v_prev+m_step*(k_v[0]+2*k_v[1]+2*k_v[2]+k_v[3])/6.0);
        m_time.push_back(t_prev+m_step);
}
```

MAIN()

```
int main()
{
        Runge4Solver system(0,1,0);
        while (system.t() < 3.15)
        {
                system.CalcStep();
                cout << "X=" << system.x() << "\tV=" << \
                system.v() << "\tT=" << system.t() << endl;
        }
        return 0;
}
```

МОЖНО ЛИ СДЕЛАТЬ БОЛЕЕ УНИВЕРСАЛЬНОЕ РЕШЕНИЕ?

- У нас жёстко задан размер системы уравнений
- Снаружи класса - какие-то непонятные функции *f1* и *f2*
- Как быть, если мы захотим сменить систему уравнений?
- *Вспомним про возможность наследования!*

```
class Runge4Solver
{
public:
        Runge4Solver(){}
        void InitValues (std::vector<double>& initVals, double initTime)
        {
                m_values = initVals;
                m_time = initTime;
        }
        void CalcStep()
        {
                ...
        }
        double t() const {return m_time;}
        std::vector<double> vals() const {return m_values;}

protected:
        virtual std::vector<double> RecalcSystem(double time) =0;
        std::vector<double> m_values;
        double m_time;
        double m_step = 0.01;
};
```

БАЗОВЫЙ КЛАСС С ВИРТУАЛЬНЫМ МЕТОДОМ

```
void CalcStep()
{
        std::vector<double> tmp,
                            partialVals(m_values.size()),
                            previousVals = m_values;
        tmp = RecalcSystem(m_time);
        for (int i = 0; i<tmp.size(); i++)
        {
                partialVals[i] = tmp[i];
                m_values[i] = previousVals[i]+tmp[i]*m_step*0.5;
        }
        tmp = RecalcSystem(m_time+m_step*0.5);
        for (int i = 0; i<tmp.size(); i++)
        {
                partialVals[i] += 2*tmp[i];
                m_values[i] = previousVals[i]+tmp[i]*m_step*0.5;
        }
        tmp = RecalcSystem(m_time+m_step*0.5);
        for (int i = 0; i<tmp.size(); i++)
        {
                partialVals[i] += 2*tmp[i];
                m_values[i] = previousVals[i]+tmp[i]*m_step;
        }
        tmp = RecalcSystem(m_time+m_step);
        for (int i = 0; i<tmp.size(); i++)
        {
                partialVals[i] += tmp[i];
                partialVals[i] *= m_step/6.0;
                m_values[i]= previousVals[i]+partialVals[i];
        }
        m_time+=m_step;
}
```

СОБСТВЕННО, ИНТЕГРИРОВАНИЕ

Здесь мы несколько раз вызываем виртуальную функцию RecalcSystem()

```
class TestRunge4 : public Runge4Solver
{
        double B1=1,
        B2=1,
        F=10;
        virtual std::vector<double> RecalcSystem(double time)
        {
                std::vector<double> ret(m_values.size());
                ret[0] = m_values[1];
                ret[1] = -B1*pow(m_values[1],2) - B2*m_values[1] + F;
                return ret;
        }
};
```

А ВОТ И КЛАСС НАСЛЕДНИК

В нём реализована та самая функция

```
class MyRunge4 : public Runge4Solver
{
        virtual std::vector<double> RecalcSystem(double time)
        {
                std::vector<double> ret(m_values.size());
                ret[0] = m_values[1];
                ret[1] = -m_values[0];
                return ret;
        }
};
```

И ЕЩЁ ОДИН

Теперь я могу создать кучу классов – под каждую систему урвнений

MAIN()

```
int main()
{
        MyRunge4 system;
        std::vector<double> init;
        init.push_back(0);
        init.push_back(1);
        system.InitValues(init, 0);
        while (system.t() < 3.15)
        {
                system.CalcStep();
                std::vector<double>vals = system.vals();
                cout << "X=" << vals[0] << "\tV=" << \

                vals[1] << "\tT=" << system.t() << endl;
        }
        return 0;
}
```

ТРЕТЬЕ ЗАДАНИЕ

- Будем решать уравнение $\frac{dy}{dt} = at - by$
- Начальное условие $y(0) = d$
- Решаем на интервале $[0,1]$ с шагом 0.01
- Аналитическое решение $\mathrm{u}(t) = \frac{a}{b}\left(t - \frac{1}{b}\right) + C \cdot e^{-bt}$
- Очевидно $-\frac{a}{b^2} + C = d$
- Нужно решить и вывести значения
- и максимальную величину рассогласования $|u(t) - y(t)|$

Вариант	a	b	d
1	1	0.2	1
2	-2	0.2	1
3	0.3	0.2	1
4	-0.7	0.7	1
5	1.2	0.7	0
6	-0.8	0.7	0
7	2.5	2.1	0
8	-2	2.1	0
9	0.3	2.1	0.5
10	-0.7	1.3	0.5
11	1.2	1.3	0.5
12	-0.8	1.3	0.5
13	2.5	1.3	0.5